

# The Anatomy of Large-Scale Distributed Graph Algorithms

Jesun Sahariar Firoz  
jsfiroz@iu.edu

Thejaka Amila Kanewala  
thejkane@iu.edu

Marcin Zalewski  
zalewski@iu.edu

Martina Barnas  
mbarnas@indiana.edu

Andrew Lumsdaine  
lums@iu.edu

Center for Research in Extreme Scale Technologies (CREST)  
Indiana University, Bloomington, IN, USA

## ABSTRACT

The increasing complexity of the software/hardware stack of modern supercomputers has resulted in an explosion in the parameter space for performance tuning. In many ways performance analysis has become an experimental science, made even more challenging due to the presence of massive irregularity and data dependency in important emerging problem areas. As with any experimental science, a characterization of experimental conditions is important for identifying which variables in the experiment affect the outcome. To gain insight into the experimental nature of performance analysis, we analyze how the existing body of research handles the particular case of distributed graph algorithms (DGAs). We distinguish *algorithm-level* contributions, often prioritized by authors, from *runtime-level* concerns that are harder to place. With a careful exposition of the tuning process for a high-performance graph algorithm, we show that the runtime is such an integral part of DGAs that experimental results are difficult to interpret and extrapolate without understanding the properties of the runtime used. We argue that in order to gain understanding about the impact of runtimes, more information needs to be gathered. To begin this process, we provide an initial set of recommendations for describing DGA results based on our analysis of the current state of the field.

## Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Concurrent Programming—*Distributed programming*

## General Terms

Performance, Algorithms

## Keywords

Runtime, Distributed, Graph, Algorithms, Performance

## 1. INTRODUCTION

Large irregular applications are gaining recognition as the future challenge in parallel computing. This is reflected by the Graph500 benchmark [22], the subject of which is the prototypical irregular problem of graph traversal. Graph traversal is a basic building blocks of other graph algorithms used in social network analytics, transportation optimization, artificial intelligence, power grids, and, in general, any problem where data consists of entities that connect and interact in irregular ways. The current Graph500 benchmark is based on breadth-first search (BFS) with a proposal to extend the benchmark with single-source shortest paths (SSSP). In this paper, we concentrate on BFS and SSSP for the same reasons, i.e., as representatives of a class of irregular graph problems.

Research on distributed graph algorithms is an emerging and active field. New algorithms, new approaches to distribute the data and new performance results appear at most major distributed computing conferences. The Graph500 benchmark bears witness to the progress, with the best results progressing from 7 GTEPS (billions of traversed edges per second) in 2010, 253 GTEPS in 2011, 15 TTEPS (trillions of TEPS) in 2012, to 23 TTEPS in 2014. Many new algorithmic techniques have been developed, e.g., direction optimization [5, 6], pruning [10], k-level asynchronous algorithm [18], hybrid algorithms [10], and distributed control [33]. A practitioner faces a multitude of published approaches, which are often vague on low-level details of implementations.

Graph problems are difficult to fit to the conventional High Performance Computing (HPC) platforms. Over the decades of development, these platforms have been optimized for problems exhibiting good locality and regular memory access and communication patterns, benefiting from caching and high-bandwidth regular collective operations. In contrast, graph algorithms exhibit little locality, rarely require any significant computation per memory access, and result in high-rate communication of small messages. Unlike regular applications that are built on top of well understood regular communication and memory access, graph algorithms interact with the whole software and hardware stack in a complex way due to their data-driven, fine-grained, irregular nature of tasks. Each piece of the stack, designed independently, from the application level, through the transport layer, to the hardware layer and the topology of the physical network, interacts within the system in unpredictable ways. This makes designing distributed graph algorithms a truly experimental science, and this state of affairs will be only exacerbated as we move towards exascale computing.

We argue that the advancements in the field are hard to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Supercomputing 2015 Austin, Texas USA

Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...\$15.00.

generalize and reconcile because the information reported is commonly incomplete. The low-level details of implementations are often vague or missing. Yet, these can have important impact. In this paper we present evidence of impact of low-level transport, scheduler, and hardware, which we refer to as *runtime*. It should be noted that the complexity of the interactions between high-level algorithm and low-level runtime that we expose is not unknown to the scientists in the field. This knowledge is implicit, fragmented, and often sidelined in presentation of new techniques. Notably, Checoni and Petrini [12], who achieve the top results in the Graph 500 benchmark in part due to direct access to the SPI (System Programming Interface) low-level primitives, provide an outstanding analysis of their evolving implementation, including a 3-years timeline of changing conclusions and understanding. However, this is not typical for the field. We point out a need for community standards and guidelines for presenting experimental results.

### Contributions:

**A motivating case study** (Sec. 2);

**A survey and analysis of the field** (Sec. 3), identifying, classifying, and discussing two levels of distributed graph algorithms: (i) Application-level aspects (Sec. 3.1) that authors identify as the main algorithmic contributions of their research; and (ii) Runtime-level aspects (Sec. 3.2) that authors do not explicitly consider a part of the algorithm but that play a crucial role in the overall performance;

**An initial set of recommendations** (Sec. 4) for authors to consider when describing their research.

## 2. MOTIVATING CASE STUDY

Pingali et al. [27] classify algorithms into two main categories: *ordered* and *unordered*. Ordered algorithms require ordering of tasks for correctness whereas unordered algorithms do not. Parallelizing ordered algorithms is challenging as parallel execution must maintain the ordering. Unordered algorithms are easier to parallelize as tasks can be executed in any order. Although the order of execution does not impact correctness in unordered algorithms, a *task priority* can be used to partially order tasks, improving performance. Our *distributed control* (*DC*) approach [33] is a work scheduling method for distributed unordered algorithms that benefit from task priority. The goal of *DC* is to remove the overhead of synchronization and global data structures by using only local knowledge to select best work, thus obtaining an approximation of the global ordering. Because *DC* does not use global data structures and synchronization, it is particularly sensitive to the runtime characteristics such as timing of task execution, communication latency, and buffering. In fact, their effect can be so significant as to make *DC* an infeasible approach if not handled carefully. In this section, we discuss the significant runtime effects we have observed.

Figure 1 illustrates *DC* for SSSP. A distributed system consists of workers divided into several shared memory domains. Each domain contains a part of the global data. Processing a task on one domain may generate more tasks that depend on data on other domains. Remote tasks are communicated through an unordered *global task bag* where every worker puts the tasks it generates. Workers continuously try to retrieve tasks from the bag into their *private working sets* that are ordered according to task priority. Because the task bag is unordered, the more tasks in the bag and not in the

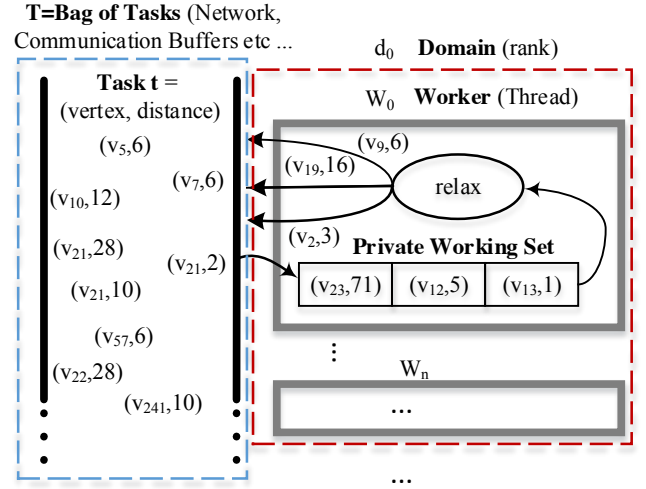


Figure 1: An overview of *Distributed Control*, using SSSP as an example.

ordered private priority queues of workers, the further the approximated ordering is from the ideal least-work ordering (Dijkstra’s priority queue).

Ideally, the underlying runtime system delivers tasks to the appropriate ordered private workset as soon as possible. On the other hand, quick delivery comes at a cost: the accumulative costs of network sends overhead, the necessity for frequent polling, frequent context switches when handling small tasks, and so on, add up to a significant overhead. To balance these competing needs, we use the AM++ [29] runtime. AM++ is particularly suitable for this purpose because it supports fine-grained parallelism of active messages with communication optimization techniques such as scalable addressing, active routing, message coalescing, message reduction, and termination detection.

In the remainder of this section, we discuss the characteristics of the runtime we experimented with and the impacts we observed. Our experiments were run at different times on different machines (Cray environments). The experiments we discuss here were ran on Big Red 2 at Indiana University [1] and on Edison at NERSC [4]. The most important differences between the two machines are the topologies, 3-D torus on Big Red 2 vs. Dragonfly on Edison, and the MPI implementations, Cray’s Message Passing Toolkit (MPT) 6.2.2 on Big Red 2 vs. MPT 7.1.1 on Edison. All experiments were run on Graph500 graphs.

### 2.1 Runtime Parameters

*DC* consists of layers, each with its own set of possible design choices and parameters. The exact set of parameters suitable for an application depends on the specifics of the machine (network overheads, etc.) and on the input (graph structure, edge weights, etc.). Some examples of parameters in our implementation include:

- Progress related parameters such as the threshold for eager progress and the frequency of progress calls.
- Optimizations such as reduction cache size, priority messaging, and “self-send” check.
- Coalescing buffers size, controlling the maximum number of messages that can be sent together.
- MPI-related parameters such as receive depth, the num-

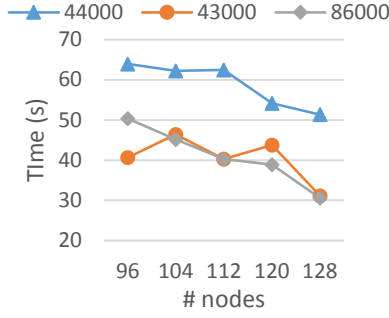


Figure 2: Effect of coalescing size on *DC* SSSP algorithm on a scale 31 graph (Big Red 2).

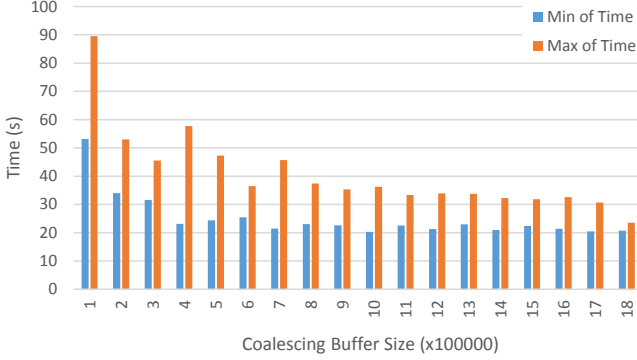


Figure 3: Effect of coalescing size on *DC* SSSP algorithm on a scale 31 graph (Edison).

ber of polling tasks, and the number of outstanding sends allowed.

We discuss some of these parameters in more detail in the remainder of this section. In addition to parameters directly related to AM++ and our algorithm, there are environment parameters, including:

- MPI progress model, with several choices on the machines that we have run on.
- Job placement, which can severely impact performance.
- Use of huge pages modules on Cray machines.

The parameter space is further enlarged by the characteristics of inputs. For example, we observed that edge weights have a significant impact on the choice of optimal parameters for *DC*.

## 2.2 Coalescing Size

To increase bandwidth utilization, AM++ performs *message coalescing*, combining multiple messages sent to the same destination into a single, larger message. Messages are appended to per-destination buffers, and to handle partially filled buffers, a periodic check is performed to check for activity. In the case of *DC* SSSP, a single message consists of a tuple of a destination vertex and distance, 12 bytes in total. With such small messages, coalescing has great impact on the performance, but finding the optimal size is difficult.

We investigated the impact of coalescing in graph500 scale 31 graphs when running *DC* SSSP with max edge weight of 100(Figs. 2 and 3). Figure 2 shows the large impact of a small change in the coalescing size, measured by the number of SSSP messages per coalescing buffer. Changing the coalescing size by less than 2% causes over 50% increase

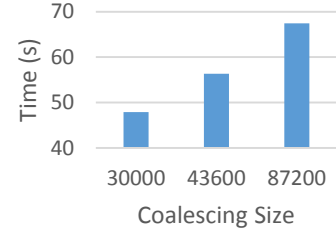


Figure 4: Effect of coalescing size on *DC* BFS algorithm on a scale 31 graph (Big Red 2).

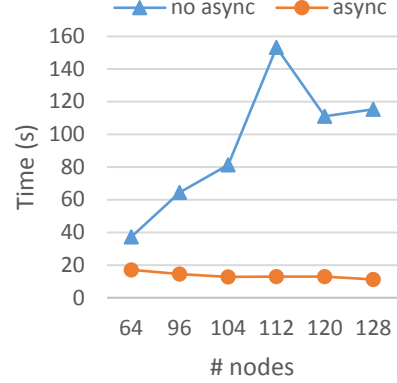


Figure 5: Effect of asynchronous progress on Big Red 2.

in the run time. This unexpected effect is caused by the specifics of Cray MPI protocols. At the smaller coalescing size, full message buffers fit into rendezvous R0 protocol that sends messages of up to 512K using one RDMA GET, while the larger buffers hit R1 protocol that sends chunks of 512K using RDMA PUT operations. At the size of 44000, the bulk of the message fits into the first 512K buffer, and the small remainder requires another RDMA PUT, causing overheads. The sizes 43000 and 86000 fill out 1 and 2 buffers, respectively, achieving similar performance. The larger size, 86000, results in better scaling properties.

We ran a more extensive suite of benchmarks on Edison. Figure 3 shows the coalescing buffer size experiments on Edison. The results are similar, with a periodic increases in the minimum run time as protocol buffers mismatch the coalescing buffers. The maximum run times signify the worst run time, as other parameters than coalescing are adjusted. The results show that adjusting other parameters is less and less important as the coalescing buffer size increases.

Figure 4 shows the effects of coalescing on a *DC* BFS, which is SSSP with maximum weight of 1. Surprisingly, increasing the coalescing size impacts performance negatively. We suspect that with smaller weights the possibility of reward from optimistic parallelism in *DC* decreases, and the added latency of coalescing has a much larger effect than with larger weights. Also note that we have not actually discovered the optimal coalescing size, which would require more experiments and more resources. All three cases shown in Figs. 2 to 4 show that adjusting the coalescing size is important, and the optimal value is not static. Rather, it depends on algorithmic concerns such as reward from optimistic parallelism.

## 2.3 Transport Progress

At first, when we experimented with *DC* on Big Red 2, we

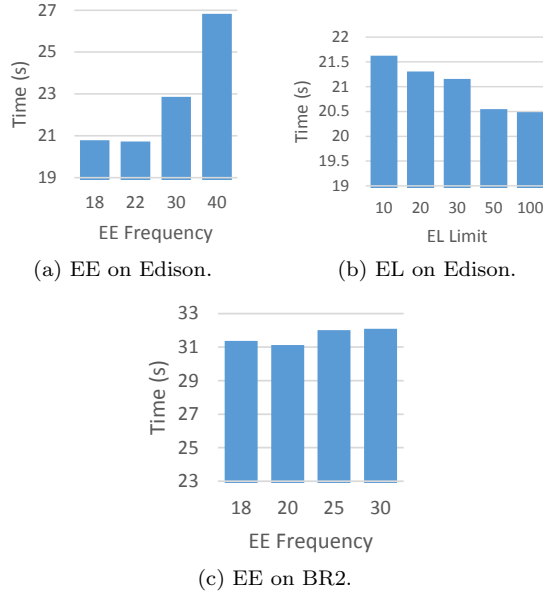


Figure 6: Effect of AM++ progress parameters.

found out that it was performing worse than  $\Delta$ -stepping. This raised a concern that the *DC* approach may not be practical. We suspected the possibility of message latencies being a culprit; so, upon researching MPT, we decided to experiment with asynchronous progress, which uses separate threads to perform progress in certain situations. Despite Cray’s warning at the time that thread-multiple progress required for asynchronous progress “is not considered a high-performance implementation”, we observed significant gains for *DC*, shown in Fig. 5. We ran the experiment on Graph500 scale 31 optimal strong scaling results. Without asynchronous progress, performance decreased with the increased number of nodes (with an unexplained anomaly at 112 nodes). (Note that all our experiments are averaged; thus, large anomalies are indicative of unexpected circumstances.) With asynchronous progress thread, the performance of *DC* has improved more than tenfold with growing node counts, entirely changing the viability of the approach. This dramatic effect illustrates how deeply an algorithm interacts with the runtime, and how a gap in parameter space may lead to incorrect conclusions about DGA approaches. Interestingly, we did not observe a similar effect on Edison, where two different asynchronous progress and the standard progress modes perform similarly.

## 2.4 Distributed Control Progress

In addition to transport layer progress, AM++ performs its own internal progress when AM++ interfaces are called. Since *DC* is built around a loop that empties the local priority data structure, it must occasionally, with some frequency call into the appropriate AM++ interfaces that perform progress. This frequency is controlled by 2 parameters: the end-epoch test frequency (EE) and the eager progress limit (EL). EE controls how many iterations of the *DC* loop run before AM++ progress is invoked. The eager limit is a threshold of outstanding *DC* tasks below which AM++ progress is performed every iteration of the *DC* loop.

Figure 6 shows the effects of progress parameters, using performance data averaged over multiple runs while varying orthogonal parameters. Edison shows a significant sensitivity

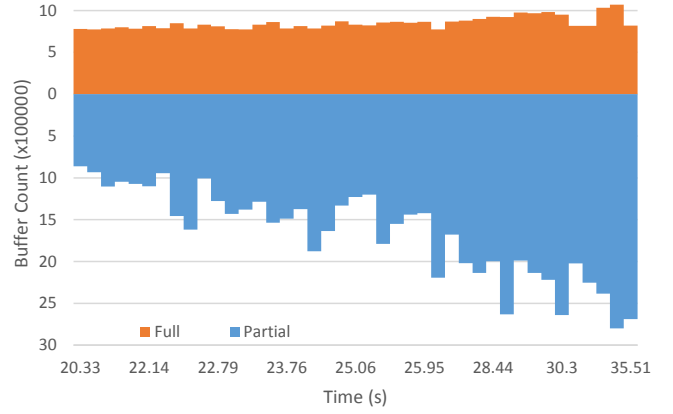


Figure 7: The impact of partial and full buffer counts on performance with coalescing size fixed at 100000 (Edison).

to the EE parameter. Smaller values are better, with 22 being the best of the ones tested. This suggests that latency may be a limiting factor on Edison. On Big Red 2, the results of varying the EE parameter are less pronounced, but the average of multiple experiments that we show here still suggests some sensitivity with the optimal value similar to that on Edison. Altogether, the results show that the performance of *DC* depends on the progress model.

## 2.5 Buffering and Work Efficiency

The prerogative of coalescing in AM++ is to decrease the overhead by sending as many full coalescing buffers as possible. Partially filled buffers are only sent when no more messages are being inserted. Figure 7 shows *DC* results on Edison for coalescing buffer size of 100000. We found that the best predictor of performance is the amount of partial buffers (fewer is better) followed by full buffers (more is better). Partial buffers indicate periods of lack of work, and this, in turn, indicates that the local priority queues are getting depleted more often, decreasing overall performance. AM++ was originally optimized for algorithms like BFS and  $\Delta$ -stepping, which benefit from eager optimization of communication overhead and are not sensitive to work imbalance. Our example shows that optimization of runtime for a seemingly worthy goal can negatively impact applications that have other needs not anticipated by runtime developers.

## 2.6 Performance Irregularity

A large supercomputer runs jobs on a complex topology, with an allocation system designed to maximize utilization and ensure fairness. To satisfy these goals and to account for complex topologies, jobs may be mapped to hardware in different ways between different runs, and hardware resources may be shared by jobs in different degrees depending on the current job configuration. Figure 8 shows scaling results for *DC* together with our implementation of  $\Delta$ -stepping for comparison. *DC* experiences a drop in performance after 96 nodes, as does  $\Delta$ -stepping, although *DC* recovers faster than  $\Delta$ -stepping. A similar drop can be observed for scale 30, but it occurs later, at 112 nodes. *DC* for scale 29 also shows a drop in performance after 104 nodes, but it is not as dramatic as at larger scales. Currently, we do not have an explanation for performance variability, but we suspect the effects of job placement as discussed in Sec. 3.2.3, where at some node counts a job on a certain input does not map

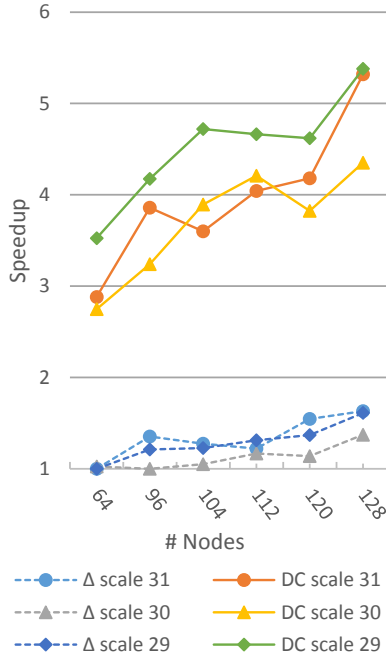


Figure 8: Irregularities in strong scaling on Big Red 2.

well to the underlying topology.

Another kind of irregularities are not as consistent, and they depend on transient conditions such as availability of contiguous blocks of nodes and interaction with other jobs. For example, on Edison, we noticed that some jobs we run more than once took up to 25% more time in some runs.

## 2.7 Caching

We implemented a write-through cache with the most recent SSSP messages. When a message to a given destination is discovered in cache, it is either discarded if the previous message had smaller distance, or it replaces the old message in cache and is sent through. Caching can improve performance dramatically, as we observed in our implementation of  $\Delta$ -stepping. However, for  $DC$ , with caching performance deteriorates. For example, on runs on scale 30 graph on Big Red 2,  $DC$  performed by almost 30% worse with caching despite reducing work by about 10%. The interesting conclusion is that an optimization technique can be actually detrimental to the performance of an algorithm, which can be counterintuitive.

## 2.8 Work vs. Overhead

Performance of an algorithm depends on the amount of *work* it performs and on the amount of *overhead* that this work incurs in a given runtime. Figure 9 shows the work statistics comprising of useful work, useless work, rejected work and invalidated work for  $DC$  and our implementation of  $\Delta$ -stepping with scale 31 graph. While  $DC$  performs better than  $\Delta$ -stepping,  $DC$  always executes more work than  $\Delta$ -stepping in the most efficient configurations of both of the algorithms. As can be seen from Figs. 8 and 9, despite consistently performing 10%-25% more work,  $DC$  performs better in all instances of tests at scale. This shows that synchronization and uneven distribution of work have an important effect on the performance of DGAs. While one can attempt to mitigate the work imbalance with algorithmic

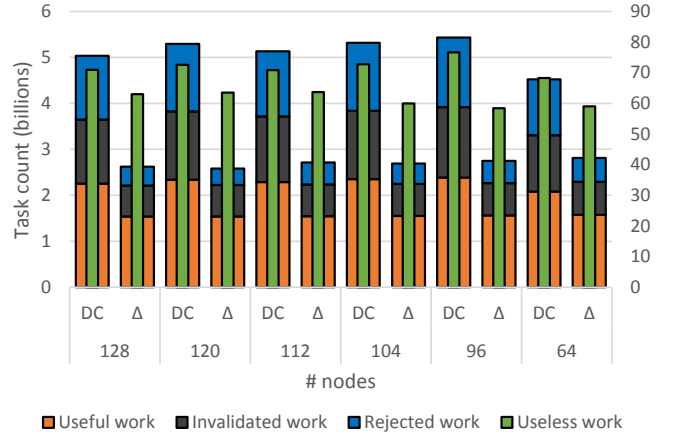


Figure 9: Proportion of different kind of works in DC-SSSP and  $\Delta$ -stepping algorithm.

techniques, the cost of synchronization is hard to control and eliminate. In this regard, an underlying runtime can have a significant impact. The more an algorithm depends on keeping global information about the runtime (e.g., for load balancing), the higher the costs of synchronization necessary to maintain that information. In Figure 9 we count a task as rejected when the vertex distance in delivers is higher than what is already recorded and, consequently, the task is not inserted into the priority queue of  $DC$  or a bucket of  $\Delta$ -stepping. Invalidated tasks are similar to rejected tasks, but their distance expires while they wait in priority queue.

## 3. THE ANATOMY OF DGAs

In previous section we show the interactions between runtime and the application cannot be neglected. Next we analyze the existing research on distributed BFS and SSSP problems. We provide *the anatomy* of DGAs, consisting of *application-level* (Sec. 3.1) and *runtime-level* (Sec. 3.2) aspects. As explained in Sec. 1, this division is based on how the research results are presented in the field. Authors usually concentrate on the application-level aspects, framing their contributions at that level, and treat the interactions with runtime as secondary results, often providing incomplete information about it. The purpose of our analysis is to describe the complexity of interactions between application and runtime and to provide a blueprint for a more complete treatment of DGAs.

### 3.1 Application-Level aspects of DGAs

Table 1 summarizes the set of parameters we identified as the application-level aspects of DGAs, and divide them into 4 categories. The *approach* category is about the main algorithmic choices, the *algorithmic considerations* category covers the main aspects of the approach, and the categories of *graph representation* and *data structures* cover the data structures that are used.

#### 3.1.1 Approach

The approaches fall into three main categories: *unordered*, *ordered*, and *mixed* (cf., Sec. 2). Unordered algorithms expose parallelism and decrease the need for synchronization.  $DC$  orders work locally, in private priority queues, and HSync does not order work at all (chaotic traversal) in its unordered



Approach		Sec. 3.1.1
Ordered	Level-Synchronous <sup>[12, 32]</sup>	
	Bellman-Ford <sup>[23]</sup>	
	Combinatorial BLAS <sup>[5, 8, 9]</sup>	
Ordered/unordered	Hybrid <sup>[10]</sup>	
	HSync <sup>[31]</sup>	
Unordered	Distributed control <sup>[33]</sup>	
	KLA <sup>[18]</sup>	
	$\Delta$ -stepping <sup>[14, 23]</sup>	
Algorithmic Considerations		Sec. 3.1.2
Data Distribution	1D <sup>[10, 12, 14, 23, 33]</sup>	
	2D <sup>[8, 9, 32]</sup>	
	Edge list <sup>[25, 26]</sup>	
Optimizations	Ghosts <sup>[16, 20, 25, 26]</sup>	
	Direction optimization <sup>[5, 9, 10, 12]</sup>	
	Pruning <sup>[10]</sup>	
	Priority messages <sup>[33]</sup>	
	Tree-based broadcast, reduction, and filtering <sup>[26]</sup>	
Load Balancing	Per-thread work splitting <sup>[10]</sup>	
	Random shuffling of vertex identifiers <sup>[8]</sup>	
	Delegates <sup>[26]</sup>	
	Proxies <sup>[10]</sup>	
Graph Representation		Sec. 3.1.3
CSR <sup>[5, 8, 9, 23, 33]</sup> , compressed coarse-index adjacency list <sup>[12]</sup> , skip list <sup>[12]</sup> , doubly-compressed sparse column (DCSC) <sup>[8, 9]</sup>		
Data Structures (algorithm progress)		Sec. 3.1.4
Distributed async visitor queue <sup>[25, 26]</sup> , thread-local priority queue <sup>[24, 33]</sup> , dynamic-array buckets <sup>[23]</sup>		

Table 1: Application-Level aspects of DGAs.

phase. Because of the lack of global ordering, both of these approaches are strongly affected by the order of task execution and message order (e.g., see Sec. 3.2.2).  $\Delta$ -stepping, KLA and Hybrid (which executes  $\Delta$ -stepping in unordered phase) utilize rounds of unordered computation separated by global barriers, where the computation is divided into buckets based on a meta property (e.g., distance in SSSP) in  $\Delta$ -stepping or on graph topology in KLA. Thanks to the use of global rounds, these approaches are less susceptible to task and message ordering, but they can exhibit *straggler effect* [33] where parts of the system are idle while other parts are still finishing the rounds. On the other hand, the ordered approaches rely on global barriers to order execution, and because of that they are susceptible to straggler effect. Algorithm load balancing (Sec. 3.1.2) and runtime load balancing (Sec. 3.2) are important characteristics in such algorithms to distribute load evenly among processes and reduce straggler effect. Ordered algorithms naturally fit the BSP model[28], but careful implementations can efficiently overlap computation and communication (e.g., [12]).

### 3.1.2 Algorithmic Considerations

**Data distribution.** There are 3 main ways to distribute graph data among the processing elements. They are *1D distribution*, *2D distribution* and *Edge List Partitioning (ELP)*.

A graph distribution makes use of certain runtime characteristics. For example, 2D distribution based algorithms com-

monly use collectives from transport (discussed in Sec. 3.2.1). 2D distributions with collectives may suffer from memory scalability issues in small memory machines. To avoid such memory scalability issues, researchers have developed collectives using point-to-point communication [32].

A graph distribution is also connected to the *logical topology* (discussed in Sec. 3.2.3) of the network, which defines the shape of the 2D distribution (square, rectangle etc.). Buluç et al. [9] showed that the logical topology of 2D distribution has an impact on the performance of the algorithm as it effects both communication and computation. They call it *Processor Grid Skewness*. In particular, “skewness” changes the number of processors in collective operation and also the size of the local data structure.

Yoo et al. [32] showed that the number of processors participating in collective communication for 1D distribution is  $O(p)$  whereas, in 2D distribution, it is reduced to  $O(\sqrt{p})$  ( $p$  is the number of processors in the process group) and claimed 2D is better for distributed BFS. But recently Checconi and Petrini [12] compared 2D distribution implemented in [13] with their 1D distribution and found that the performance of the BFS implementation with 2D distribution is slower compared to 1D because the 2D implementation did not employ runtime characteristics such as compression, communication and computation overlapping methods. Further they also showed scaling was limited in 2D due to cache thrashing.

To overcome certain overhead incurred due to high degree vertices (in 1D), Pearce et al. [25] introduced ELP. For ELP, we sort the edgelist of a graph according to their sources and partition them evenly. But with ELP, a vertex state may reside in multiple nodes (see Delegates under Load Balancing) and requires synchronization of states between nodes via reductions.

**Optimizations.** Research community in this area uses various techniques to improve the efficiency of a proposed algorithm.

Beamer et al. [5] introduced direction optimization to standard level synchronous BFS algorithm. Direction optimization reduces the number of vertices to be visited by traversing vertices in a bottom up fashion. Beamer et al. [5] showed that direction optimization reduces contention for some atomic operations as bottom up approach removes the need for atomic operations (because only child writes to itself, hence removing the contention). Though direction optimization is a promising approach to improve performance, the initial method suffered from several runtime difficulties [6]. Each processor needs to check the membership of the frontier set but frontier set is too large to replicate across nodes. In addition, each vertex requires searching for its parent sequentially. To solve the first problem authors of [6] used 2D distribution and to tackle the second problem they used systolic shifts.

High degree vertices are challenging for distributed graph algorithms. *Ghost vertices* is an optimization technique to alleviate overhead incurred by high degree vertices. Ghost vertices reduce some remote communication by locally storing the distributed state of high degree vertices. For larger graphs, ghost vertices can limit the memory scalability.

In SSSP, when an algorithm detects a vertex with relatively lower distance the processing node can relax that vertex with high priority. Messages generated using such vertices are called *priority messages*. During the execution of an algorithm, if a node observes that a message with high

priority (e.g., better vertex distance in SSSP) is received, it is processed immediately bypassing other intermediate data structures. Runtime support is necessary for this optimization to be successful. The priority messages should get delivered to respective owners swiftly. Therefore the sender needs to maintain a separate channel for priority messages. Further if runtime is supporting message aggregation (coalescing), the priority messages should maintain low coalescing buffer size compared to normal buffers.

**Load Balancing.** Load balancing in general attempts to distribute work among participating processes uniformly. In the following we discuss few strategies for load balancing.

*Per thread work splitting & Proxies:* Chakaravarthy et al. [10] used two tier mechanism: intra-node thread level and inter-node vertex-splitting strategies, based on proxies, to achieve load balancing. In intra-node thread level load balancing, besides the owner thread of a high degree vertex, other threads can participate in the relax operation for the edges involved. In-node NUMA characteristics affect intra-node load balancing. For inter-node processing, the authors employed *proxies*. Proxies reside in different localities than the high degree vertex. Proxies are connected to the high degree vertex with 0 weight. Time for processing a high degree vertex connected with proxies depends on the job placement, i.e. if a proxy is connected through a slow channel, the time to process the vertex relevant to the proxy is increased.

*Random shuffling of vertex identifier:* Buluç and Madduri [8] achieved a reasonable load balancing by randomly shuffling all the vertex identifiers prior to partition. In the runtime, random shuffling changes the algorithm communication pattern. This helps to share runtime resources near equally among all processes.

*Delegates:* Pearce et al. [26] used *delegates* to distribute edge lists of high-degree vertices across multiple processes. While partitioning, the outgoing edges are placed at the edge’s target vertex location. Then one of high degree vertex owners is designated as a *controller* and others are assigned as *delegates*. The controller maintains the state of the vertex and delegates keeps a copy of the updated state. Delegates communicate with each other using asynchronous broadcast and reduction operations. According to the authors of [26], distributed delegate technique is more efficient compared to ELP in terms of communication reduction.

### 3.1.3 Graph Representation

In the literature we find two main techniques to represent a graph. They are *Adjacency list* and *Compressed sparse row (CSR)*. The CSR representation minimizes the space needed using a row indexing mechanism. Compared to adjacency list representation, CSR is a more compact and efficient data structure but less versatile.

Edmonds et al. [14] showed that, due to compact size (less memory compared to adjacency list) and efficient access methods (direct indexing), the CSR implementation outperforms adjacency list representation. Very recently, Buluç et al. [9] showed that, at large scales (33 and above), the CSR representation does not scale with memory. Therefore to represent larger graphs, [9] used *Doubly Compressed Sparse Column (DCSC)*. As per the authors, CSR representation is faster than DCSC. They also demonstrated that CSR performance is affected by in-node multithreading and NUMA behaviors. The authors received 15-17% performance improvement when executed with multi-threading within NUMA domains with

the CSR implementation.

Storing adjacencies in linked lists incurs cache misses on traversal (pointer chasing of list nodes). To avoid sequential navigation Checconi and Petrini [12] used *indexed skip lists* for adjacency lists. Skip lists contain shortcuts to navigate from one source vertex to the following. When a few vertices need to be visited, a coarse index allows to skip large portions of adjacency list. However Skip Lists need more memory to store additional indexing data (compared to standard adjacency list representation).

### 3.1.4 Data Structures for Algorithm Progress

Algorithms use data structures to maintain intermediate state. Pearce et al. [25] used a *visitor queue* that is implemented as a collection of priority queues, to maintain adjacency vertices of an already visited vertex. A priority queue for a vertex is selected based on a hash function. Using multiple threads with a hash function reduced lock contention. We used *thread local priority queues* for DC SSSP [33]. We took this approach to avoid contention on priority queues. To represent frontier vector for BFS, Buluç and Madduri [8] used a thread-local stack. These are merged into frontier set at the end of each iteration. Though this approach reduces contention, it needs additional memory to occupy thread-local stacks and copying also takes time. In our  $\Delta$ -stepping [33] implementation we used a shared memory bucket data structure that is based on arrays. Atomics were used to avoid any race conditions when making changes to buckets.

## 3.2 Runtime-Level aspects of DGAs

In Sec. 2, we show how different runtime-level parameters effect the performance of DGAs. In this section, we categorize important runtime-level parameters based on the review of existing literature for DGAs. Table 2 summarizes the set of low-level runtime parameters.

### 3.2.1 Communication Paradigm

The choice of *communication paradigm* can have a notable impact on performance of DGAs. Each paradigm imposes different tradeoffs in terms of memory constraints, synchronization overhead and network latency. The *collectives* paradigm is used when large low-overhead stages of all-to-all communication are needed, *point-to-point* paradigm allows for finer overlap between computation and communication at the expense of code complexity, and *active messages* are a refinement of point-point communication that adds an implicit execution of *handlers* on remote objects. Finally, *one-sided* paradigm provides remote memory operations (GET, PUT, etc.) which are very efficient, but require remote memory management protocol. For example, collectives are the base of BLAS approaches and level-synchronous approaches. However, Checconi and Petrini [12] show how using lightweight point-to-point communication may lead to improvements in traditionally synchronous approaches. They compare their point-to-point implementation using low level SPI interface (discussed in Sec. 3.2.2) to an MPI implementation using collectives. They note the large memory footprint required for collective buffers, which forces them to decrease the scale of the problem per node. Furthermore, collectives do not allow for easy interleaving of computation with communication. Given these two factors and the overhead of MPI over SPI, they note a fivefold decrease in performance. Active messages

<b>Communication Paradigm</b>		Sec. 3.2.1
Point-to-Point <sup>[12, 33]</sup> , Collectives <sup>[8, 9, 12, 23]</sup>		
One-Sided, Active messages <sup>[10, 33]</sup>		
<b>Transport</b>		Sec. 3.2.2
Request Tracking	Remotely synchronous Locally synchronous <sup>[12]</sup> Asynchronous <sup>[33]</sup>	
Progression <sup>[12, 33]</sup> :		
• Asynchronous	System threads <sup>[33]</sup> User threads	
• Synchronous	Explicit progress <sup>[33]</sup> Runtime scheduler <sup>[33]</sup> Lightweight task <sup>[33]</sup>	
Bit Transport	System Processing Interface (SPI) <sup>[10, 12, 13]</sup> Message Passing Interface (MPI) <sup>[8, 9, 12]</sup> Active Pebbles (AM++) <sup>[30]</sup> ARMI <sup>[17, 18]</sup>	
Protocol	Eager, rendezvous <sup>[33]</sup> , completion <sup>[10, 12]</sup>	
Optimization	Message reduction/caching <sup>[23, 26, 33]</sup> Message coalescing <sup>[12, 25, 26]</sup> Message compression <sup>[12]</sup>	
Message routing	2D, 3D <sup>[25, 26]</sup> , ring <sup>[32]</sup> , hypercube, rook <sup>[15]</sup>	
Threading	Multi-threaded <sup>[33]</sup> , serialized, funneled	
<b>Network Topology</b>		Sec. 3.2.3
Physical Topology	3D <sup>[11, 25]</sup> and 5D <sup>[11]</sup> torus, Dragonfly <sup>[9, 33]</sup>	
Logical Topology	Skewness <sup>[9]</sup> , synthetic network <sup>[25, 26]</sup>	
Job topology	Job allocation, rank mapping <sup>[9]</sup>	
<b>Local Scheduling</b>		Sec. 3.2.4
Heavyweight	Pthreads <sup>[33]</sup> , OpenMP <sup>[8, 9]</sup>	
Lightweight	Work stealing, FIFO tasks <sup>[33]</sup>	
Termination	Quiescence detection <sup>[25]</sup> , SKR <sup>[33]</sup>	
Hardware effects	L2 atomics <sup>[10]</sup> , NUMA effects <sup>[9]</sup>	
<b>Runtime Feedback</b>		Sec. 3.2.5
Optimal K-level <sup>[18]</sup> , optimal $\Delta$ , sync to async switching <sup>[31]</sup>		

Table 2: Runtime-Level aspects of DGAs.

are based on point-to-point communication, and they display similar communication performance characteristics (indeed, [12] implements rudimentary active messages using low-level system interfaces). However, AM++ provides a full scale of active message services, such as message routing, message reduction, and object-based addressing, and automatic execution of handlers [29], which requires a runtime scheduler which may have an important impact on the performance of a DGA (Sec. 2).

### 3.2.2 Transport

The transport layer is the part of the stack responsible for sending and receiving bits. Important properties of transport include how message buffers are handled, which entity manages them, and how frequently they need to be managed. The runtime needs to take several decisions regarding these.

**Request Tracking (RT)**, refers to how communication requests are made into transport: a request is scheduled and needs to be completed later (*asynchronous*), a request is made and the requester waits until all local data structures can be reused (*locally synchronous*), or a request is made and

the requester waits until it has been completely processed (*remotely synchronous*). As an example, *remotely synchronous* request tracking in MPI (e.g., by MPLSend etc.) guarantees a small number of messages on the network, but it hinders parallelization. On the other hand, using *locally synchronous* may allow more parallelism if the underlying implementation uses an eager protocol (e.g., MPLSend). Finally, *asynchronous* RT uses interfaces such as MPLIsend/MPLIrecv to start requests along with MPLTestsome to check for their completion. maximizing overlap between computation and communication. However, with the asynchronous RT, the client of a transport must make decisions about how many requests to keep opened at any given time, how many to check, at any given time. Checconi and Petrini [12] used asynchronous RT, maintaining one buffer per destination. A send operation queues messages on these buffers until the buffer is full, and then hands it over to the network interface. But before starting to write to the buffer again, their implementation has to wait for the previous send to complete so the buffer becomes available. In [33], AM++ spawns a constant number of receive requests and as many send requests as necessary, all stored and tracked in one array of MPI requests.

**Progression.** Completing a round trip through transport requires a protocol for dedicating resources to the transport for bookkeeping, performing bit moving, and delivering the results of completed requests to the application—we call that protocol progression. Progression influences the timeliness and efficiency of transport delivery, and a wrong progression model can render an application infeasible (Sec. 2.4). In *asynchronous* progression, computing resources are dedicated to make progress. The resources can be dedicated through *system* or *user* threads. For example, Cray MPI provides an option for starting progression pthreads that perform internal MPI progress in parallel with the application threads. *User threads* serve similar purpose but are started by the user explicitly, and, for example, call MPI repeatedly to generate progress. In contrast, *synchronous* progression is done periodically in the runtime or application level. In *explicit* asynchronous progress, the application can choose explicitly, bypassing the runtime scheduler (if any), when to call progress, enabling optimizations at the cost of complexity. For example, we employed explicit polling for our *DC*, but we observed a decrease in performance. In a task-based system, network progress can be scheduled as a *lightweight* task. For example, AM++ implements network polling, buffer flushing, checking for termination, and executing pending handlers for received messages as tasks, on equal footing with application task that run message handlers. Finally, a runtime with a scheduler, such as AM++, can perform progress directly in the *scheduler*, which allows for more control and runtime feedback when performing progression. Most papers do not discuss progression and request tracking explicitly, but the choices made for these parameters may have a profound effect on performance (cf., Sec. 2.4 for a motivating example).

We illustrate the progression and request tracking by briefly comparing our *DC* in AM++ with the work of Checconi and Petrini [12]. Checconi and Petrini used a lightweight asynchronous communication layer on top of System Processing Interface (SPI), with separate FIFOs for injections and receptions (up to 16 FIFOs each per node, providing network-level parallelism). They queue messages to per destination buffers



where each buffer is exclusively owned and operated on by a single thread, eliminating locking and contention. Buffers are placed into injection queues when ready, and a thread will wait for completion when another message needs to be sent to a given destination. AM++ also maintains per-destination buffers. The buffers are shared and require atomic operations for writing. Compared to Checconi and Petrini, AM++ does not wait for the send buffers to become available. Instead, it creates new buffers and can spawn multiple asynchronous send requests for the same destination. AM++ supports three different progression models. Polling can be invoked explicitly from the algorithm. Explicit polling bypasses the AM++ task queue, and directly queries the outstanding send and receive requests (with `MPI_Testsome` on an array of requests). AM++ also has special purpose user-level tasks that perform progression. These tasks are executed from AM++ task queue when sends are performed, or when end-of-epoch tests are performed (during these tests AM++ tries to finish an epoch by processing remaining work).

**Bit transport.** Bit transport is the lowest-level network interface used by upper levels to deliver bits from one location to another. In the work from the IBM group [10, 12, 13], the *System Processing Interface (SPI)* communication layer serves as a bit transport (as described above). The majority of implementations we have discussed use *Message passing Interface (MPI)* for their bit transport. SPI is a direct interface to hardware queues, while MPI is a complex framework with extra functionality and semantics. Direct interfaces like SPI may yield more efficient communication, but are less or not at all portable, and may require more implementation effort to implement higher-level features. The third type of bit transport is based on remote method invocation (RMI) technique and is used in approaches based on *STAPL* [17, 18], a generic parallel library for graph and other data structures and algorithms. STAPL uses the ARMI (Adaptive Remote Method Invocation) active-message communication library, based on RMI. ARMI supports automatic message coalescing but does not provide routing or message reductions natively. Both AM++ and ARMI can use different backends, so from the application’s perspective they are bit transport themselves, but internally they use a lower-level bit transport. Such layering makes it hard to optimize parameters related to bit transport (e.g., choosing proper coalescing size for AM++).

**Protocol.** Bit transport may employ different protocols for different message sizes. For example, MPI point-to-point communication, may support *eager* protocol for small messages and *rendezvous* protocol for larger transfers, sending messages without or with, respectively, round-trip communication [3]. The autonomous and transparent choice of protocols may have a detrimental impact on applications (e.g., Sec. 2.2).

**Optimization.** A number of runtime-level optimization techniques have been proposed in the literature to reduce communication overhead and maximize throughput. In Sec. 2.7, we discussed *message reduction* (caching) in AM++. Pearce et al. [26] used tree based broadcast, reduction and filtering for communication involving high degree vertices. This essentially forces the visitors to traverse the delegate (cf., Sec. 3.1.2) tree and provides the opportunity to filter out messages. Panitanarak and Madduri [23] used local lookup arrays to track the tentative distance of every vertex, thus avoiding duplicate request being sent.

Increasing *message coalescing* (cf., Sec. 2.2) size increases the rate at which small messages can be sent over a network at the cost of latency. Checconi and Petrini [12] used coalescing to pack together all the edges that would be sent to each destination separately and queued them in an intermediate buffer. Pearce et al. [25, 26] combined coalescing with routing to reduce dense communication. Based on the observation that bisection bandwidth becomes a limiting factor for DGAs, Checconi and Petrini [12] utilized the available processing power to *compress* their buffers of coalesced messages, using differential encoding scheme for compression of vertices. They reported that compression decreased the number of messages sent in intermediate BFS levels where high message traffic is present, but when the message traffic is low, compression degrades the performance.

**Message Routing.** Pearce et al. [25] implemented routing through a synthetic network to mimic the BG/P 3D torus interconnect topology. In a followup paper [26], the authors additionally embedded the delegate tree as a means for further communication reduction. AM++ [29] also supports two types of software routing strategies: *Rook routing* and *Hypercube routing*. Rook routing reduces the number of communicating buffers to  $O(\sqrt{p})$  [15]. However, a disadvantage of software routing is that it increases message latency. Yoo et al. [32] used ring communication in their optimized collective implementation and adjusted the diameter of the ring to achieve better performance.

**Threading.** A message passing framework can support different level of thread safety. For MPI, there are 4 levels in total [2]: *Single*, *Funneled*, *Serialized* and *Multiple*. In our DC implementation [33], we used *Multiple* as the threading level with the aim to have the maximum flexibility for multiple threads to invoke MPI functions concurrently.

### 3.2.3 Network Topology

Computing resources are organized in several specific *physical topologies*: 3D Torus, Dragon Fly, 5D Torus etc. The physical topology has a connection to collectives. For example, MPICH2 provides an all-to-all implementation that is optimized for Aries and Gemini systems. In [11], the authors stated that the triangle of virtual processors are embedded in the Blue Gene/Q physical topology which is a 5D torus.

Some of the graph traversal algorithms with 2D distribution make use of the pattern of communication between ranks and that pattern is called *logical topology*. In Sec. 3.1.2 we discussed *Processor Grid Skewness*. In relation to skewness, Buluç et al. [9] found out that the “tall skinny” grids performed faster and “short fat” grids performed worse than square grids. Pearce et al. [25] used a 3D routing topology to mirror the Blue Gene /P 3D torus interconnect topology. In this way they created a *synthetic network* to implement routing and aggregation.

Job scheduler for computing resources allocate nodes based on scheduling policy and the input request. This formulates the *job topology*. Bhatele et al. [7] showed that performance of an application depends on job placement. Specially in Cray systems the variation can be significant. The variations can be due to the distances among allocated nodes or due to contention on shared network.

### 3.2.4 Local Scheduling

Depending on the node-level threading mechanism, thread scheduling policies and synchronization primitives, tasks

associated with a DGA can execute in different order with varying frequencies. For example, in an attempt to quickly spread good work, we can send a message with priority and put the message handler in front of the task queue. This is one way to achieve priority scheduling [33]. Data structures support (for example bitmaps in sync mode and global queue in async mode in [31]) is also an important factor to achieve effective local scheduling. Below we discuss several thread-granularity and scheduling related factors.

*Heavyweight threads (worker threads)* can be used for intra-node threading. Buluç and Madduri [8] used MPI for inter node processing and GNU OpenMP for intra-node threading. Buluç et al. [9] also used OpenMP threading in their implementation as it is beneficial for algorithms implemented using BLAS. But *DC* based unordered algorithms need more control over threading. Therefore, [33] used a combination of MPI and pthreads.

Lightweight threads, implemented on top of kernel threads, can be scheduled differently. *Lightweight thread scheduling* mechanisms achieve load balancing mostly by *work stealing* and *FIFO* scheduler for tasks.

The frequency of *termination detection* (TD) is also a very important catalyst for DGA performance. This is especially true for unordered algorithms. In AM++ [33], the termination detection is implemented in the runtime level with the help of non-blocking collectives and work balancing FIFO queue. Hribar et al. [19] implemented TD in algorithm level and advocated that, when the computation time becomes smaller than the time it takes to receive a message, high detection frequency should be used. Otherwise low detection frequency should be used.

*Hardware Effects* such as NUMA, L2 atomics impact on the in-node execution of distributed graph algorithms. Chakravarthy et al. [10] exploited atomic operations implemented in the L2 cache to achieve better aggregate update rate in relaxation. [8, 9, 21] also used atomic updates to mitigate synchronization costs. Also, Buluç et al. [9] considered the effect of multithreading within NUMA domains and observed noteworthy performance gain compared to Flat-MPI runs.

### 3.2.5 Runtime Feedback

Choosing algorithmic parameters adaptively, switching between different algorithms during execution and choosing the mode of algorithms (sync vs async) depend heavily on the feedback provided by the runtime. For example, in [18], to adaptively determine and set the level of asynchrony,  $k$ , in each superstep, a set of conditions are being evaluated to take the decision about whether to double the size of  $k$  or not. One of the conditions checks whether the penalty for asynchrony (assessed in terms of wasted work) has exceeded a threshold or not. Based on the support of runtime, effect/propagation of wasted work can be mitigated by propagating better work from lower level and thus invalidating wasted work even before processing them in the algorithm level. If this scheme can be put into place in runtime, we can expect different adapted value of  $k$  compared to the one without the scheme. Similarly, Xie et al. [31] used throughput ie. the amount of vertices processed per unit time as a metric to switch between sync and async mode of algorithms. But the throughput depends on how efficiently the underlying runtime is being employed.

## 4. CONCLUSIONS

We demonstrate clearly and explicitly that the application-level parts that are reported as major contributions do not constitute a complete description of a DGA. We show how sometimes small changes in runtime can threaten the viability of an approach. We aim to raise awareness of the importance of runtime. With this in mind, we first provide a representative overview of application-level aspects in BFS and SSSP. Then we carefully analyze runtime aspects, which are usually overlooked. Based on our analysis, we provide the “anatomy of DGAs”. The anatomy consists of two major layers: the application-level aspects and the runtime-level aspects, which respectively represent the top and the bottom of the software/hardware stack. Each is further subdivided into categories, and we provide examples from existing research. We propose a set of guidelines for reporting research design and results. Altogether, the goal is to make research results in DGAs more accessible, general, and congruent. To achieve this goal, we believe that research has to be presented in the context of the whole complex stack on modern supercomputers (getting more complex with progress toward exascale). We intend the guidelines to serve the community as an initial step to iterate and expand on.

Our Tables 1 and 2 serve as an initial map for **reporting the design features**. It is as important to state which parts of DGAs anatomy are explicitly covered in the results as which are not. Some may remain “buried in the stack”, their impact unknown (for example, the effects of job placement as in Sec. 3.2.3 are not usually investigated), and some may not be relevant in a given situation. Our anatomy helps both consumers and authors of research, the former to understand and the latter to present contributions.

Next, it is important to outline the **experimental protocol** used to obtain the results. This amounts to stating which parts of the parameter space are covered by experiments and, as importantly, which are not. Furthermore, it is helpful for the reader to understand *why* certain parts of the parameter space are covered and others are not, even if the reason is as mundane as limited resources (burning through time allocations is easy). Also, it is often helpful to present negative experimental results if any were observed as they help to uncover to which parameters a given approach is sensitive.

Our analysis and guidelines are intended to be only the first, imperfect step in unifying the field. We posit that the DGA research community should collectively develop a set of standards expected from top notch research, acknowledging that DGAs exhibit particularly strong interaction with the software/hardware stack due to their irregularity. Thus we appeal to the wider community to take our initial suggestion and to help develop standards for more explicit incorporation of runtime interactions in future research results.

## 5. ACKNOWLEDGMENTS

This research used resources of NERSC (a DOE Office of Science User Facility Office of Science and U.S. Department of Energy under Contract No. DE-AC02-05CH11231) and Big Red2 (Funded by Lilly Endowment, Inc. and Indiana METACyt Initiative). Research is supported by NSF grant 1111888 and Department of Energy award DE-SC0008809.

## References

- [1] Big Red II at Indiana University. <http://rt.uits.iu.edu/ci/systems/BRII.php#info>. Accessed: 2015-04-17.
- [2] MPI\_Init\_thread. [http://www.mpich.org/static/docs/v3.1/www3/MPI\\_Init\\_thread.html](http://www.mpich.org/static/docs/v3.1/www3/MPI_Init_thread.html), . Accessed: 2015-04-10.
- [3] MPI Performance Topics. [https://computing.llnl.gov/tutorials/mpi\\_performance](https://computing.llnl.gov/tutorials/mpi_performance), . Accessed: 2015-04-10.
- [4] NERSC's Edison. <https://www.nersc.gov/users/computational-systems/edison/configuration/>. Accessed: 2015-04-17.
- [5] S. Beamer, K. Asanović, and D. Patterson. Direction-Optimizing Breadth-First Search. *Scientific Programming*, 21(3-4):137–148, 2013.
- [6] S. Beamer, A. Buluç, K. Asanović, and D. Patterson. Distributed Memory Breadth-First Search Revisited: Enabling Bottom-Up Search. In *International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pages 1618–1627, May 2013.
- [7] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs. There goes the neighborhood: Performance degradation due to nearby jobs. In *Proc. Internat. Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 41:1–41:12. ACM, 2013.
- [8] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *Proc. Internat. Conf. for High Performance Computing, Networking, Storage and Analysis*, page 65. ACM, 2011.
- [9] A. Buluç, S. Beamer, K. Madduri, K. Asanović, and D. Patterson. Distributed-Memory Breadth-First Search on Massive Graphs. In *Parallel Graph Algorithms*, D. Bader (editor), CRC Press. 2015. To appear.
- [10] V. T. Chakaravarthy, F. Checconi, F. Petrini, and Y. Sabharwal. Scalable Single Source Shortest Path Algorithms for Massively Parallel Systems. In *28th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2014)*, 2014.
- [11] F. Checconi and F. Petrini. Massive data analytics: The Graph 500 on IBM Blue Gene/Q. *IBM Journal of Research and Development*, 57(1/2):10:1–10:11, Jan. 2013.
- [12] F. Checconi and F. Petrini. Traversing Trillions of Edges in Real Time: Graph Exploration on Large-Scale Parallel Machines. In *Proc. 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 425–434. IEEE, 2014.
- [13] F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A. R. Choudhury, and Y. Sabharwal. Breaking the speed and scalability barriers for graph exploration on distributed-memory machines. In *Proc. Internat. Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 13:1–13:12. IEEE, 2012. Cited by 0023.
- [14] N. Edmonds, A. Breuer, D. Gregor, and A. Lumsdaine. Single-Source Shortest Paths with the Parallel Boost Graph Library. In *The Ninth DIMACS Implementation Challenge: The Shortest Path Problem*, Nov. 2006.
- [15] N. Edmonds, J. Willcock, and A. Lumsdaine. Expressing Graph Algorithms Using Generalized Active Messages. In *Proc. 27th International ACM Conference on International Conference on Supercomputing*, pages 283–292. ACM, 2013.
- [16] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*, volume 12, page 2, 2012.
- [17] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. The STAPL Parallel Graph Library. In H. Kasahara and K. Kimura, editors, *Languages and Compilers for Parallel Computing*, number 7760 in LNCS, pages 46–60. Springer Berlin Heidelberg, 2013.
- [18] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. KLA: A New Algorithmic Paradigm for Parallel Graph Computations. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pages 27–38. ACM, 2014.
- [19] M. R. Hribar, V. E. Taylor, and D. E. Boyce. Termination Detection for Parallel Shortest Path Algorithms. *Journal of Parallel and Distributed Computing*, 55(2):153–165, Dec. 1998.
- [20] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in The Cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.
- [21] K. Madduri, D. Bader, J. Berry, and J. Crobak. An Experimental Study of a Parallel Shortest Path Algorithm for Solving Large-Scale Graph Instances. In *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 23–35. SIAM, Jan. 2007.
- [22] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500 benchmark. *Cray User's Group (CUG)*, 2010.
- [23] T. Panitanarak and K. Madduri. Performance Analysis of Single-source Shortest Path Algorithms on Distributed-memory System. In *Proc. Sixth SIAM Workshop on Combinatorial Scientific Computing*, page 60, 2014.
- [24] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *Proc. Internat. Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2010.
- [25] R. Pearce, M. Gokhale, and N. M. Amato. Scaling Techniques for Massive Scale-Free Graphs in Distributed (External) Memory. In *Proc. 27th IEEE International Symposium on Parallel and Distributed Processing*, Los Alamitos, CA, USA, 2013. IEEE.
- [26] R. Pearce, M. Gokhale, and N. M. Amato. Faster Parallel Traversal of Scale Free Graphs at Extreme Scale with Vertex Delegates. In *Proc. Internat. Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 549–559. IEEE, 2014.
- [27] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, et al. The Tao of Parallelism in Algorithms. *ACM SIGPLAN Notices*, 46(6):12–25, 2011.

- [28] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.
- [29] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine. AM++: A Generalized Active Message Framework. In *Proce. 19th Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 401–410. ACM, 2010.
- [30] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine. Active pebbles: a programming model for highly parallel fine-grained data-driven computations. In *Proc. 16th ACM symposium on Principles and practice of parallel programming*, pages 305–306. ACM, 2011.
- [31] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen. SYNC or ASYNC: Time to Fuse for Distributed Graph-parallel Computation. In *Proc. 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2015.
- [32] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. In *Proc. Internat. Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 25–25, Nov. 2005.
- [33] M. Zalewski, T. A. Kanewala, J. S. Firoz, and A. Lumsdaine. Distributed Control: Priority Scheduling for Single Source Shortest Paths Without Synchronization. In *Proc. of the Fourth Workshop on Irregular Applications: Architectures and Algorithms*, pages 17–24. IEEE, 2014.